Enhanced Decoupling of Components in Intelligent Realtime Interactive Systems using Ontologies

Dennis Wiebusch* Marc Erich Latoschik†

Human-Computer Interaction
University of Würzburg

Abstract

We introduce a technique to support decoupling in component-based, modular software architectures as a means to enhance non-functional requirements, i.e., to increase reusability, portability, and adaptability. The core idea utilizes a semantic description of interfaces and component interplay in the area of Intelligent Realtime Interactive Systems (IRIS). Semantic descriptions are encoded as OWL-based models, which build a Knowledge Representation Layer (KRL) of relevant interface constructs and component features. These models are automatically transformed into programming language code of a given target language. The result of that transformation forms a semantically grounded database of relevant system aspects that programmers can use to develop their application. Examples, taken from an application that was developed with the Simulator X framework, illustrate the different aspects of the proposed method and demonstrate its practicability.

Index Terms: D.2.11 [Software Engineering]: Software Architectures—Domain-specific architectures I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Virtual Reality

1 Introduction

Realtime Interactive Systems (RISs) in areas like Virtual Reality (VR), Augmented Reality, or computer games are often characterized by a multitude of functional and non-functional requirements. The functional requirements reflect the various aspects of state-of-the-art interactive environments. This typically includes support for a wide variety of input/output devices and interaction methods, high quality graphics and sound rendering, physics and haptics simulation, as well as behavior simulation and Artificial Intelligence (AI) to name just the common ones.

Interactivity is the primary non-functional run-time requirement. Reusability, portability, and adaptability are of equal importance during development time. Implementation is often split between different development groups with respect to certain development skills. Core-system developers are responsible for the underlying central architecture and its provided development paradigms. Component developers are specialists for a certain functional requirement, which they want to integrate into an application, e.g., graphics or AI specialists. Content creators (or game designers) want to utilize all functional features provided, to realize their target application. RIS-programming skills may vary significantly between these groups. The design of appropriate interfaces which provide the respective functionality to each group becomes crucial. These interfaces should support scalability, operability, and usability with respect to a development view.

*e-mail: dennis.wiebusch@uni-wuerzburg.de
†e-mail: marc.latoschik@uni-wuerzburg.de

From a technical point of view, interfaces between components may become quite complicated. For example, behavior or AI components often need to access the objects residing in the virtual world at certain times during the simulation loop. Access times might not be pre-defined by a fixed schedule but might be based on possibly concurrent events. Accessing those objects should ideally be performed in the same way as done by regular components, to avoid programming overhead. Since variable and function names are generally inaccessible to the AI component, semantic reflection [29] would be necessary to achieve that goal.

From a different point of view, a generalized interface is desirable as well: As the hardware environment is a subject to change, every once in a while it may become necessary to replace components to keep the system up to date. Unfortunately, in many cases new hardware is incompatible to the software used before. Hence, to render expensive reimplementation unnecessary, modularization is a very important aspect of a RIS.

Apparently, the ideal case would be to have a single general interface, which connects all aspects of a RIS. Due to the diversity of components such a system may consist of, this interface is required to be highly flexible. For example, an Intelligent Virtual Environment (IVE) [2] will probably be composed of a rendering component, a physics engine, a user interface, and an AI component. If the rendering component should be replaced by, e.g., a ray-tracer, this will usually necessitate considerable effort. However, if all components would make use of the same interface, each component could be easily replaced. Furthermore, new components could be added without any changes to the existing system.

2 Related Work

Recent work discusses the value of decoupling for computer science [10] and especially for RIS-development [31].

The task to provide interfaces and description languages in order to decouple parts of RIS-frameworks has been tackled by various groups of researchers. Early approaches focused on specific aspects of a framework. A very important primary requirement is independency of input devices, e.g., by providing abstract software interfaces [5, 7, 26] and protocols or description languages [13, 44]. Other groups focused on creating distributed environments [15, 16, 35] and knowledge based systems [24, 29, 34].

Several techniques commonly applied to RIS-developments support decoupling in various flavors, e.g., data-flow oriented architectures [1, 41, 45], shared databases [7, 15], remote procedure calls [16], event propagation [7, 25], and integration of KRLs [9, 29, 34].

More recent work points out that a common data format [3] and the automatic conversion from and to such a format [33] are desirable features. Simulator X [32], the RIS-software utilized for the work described here, provides a virtual shared database as well as an event propagation system and supports a common data format and automatic data type conversion, which is detailed in section 3.

Further important aspects of RIS-development are content description techniques, used to define and access elements of virtual environments (VEs). A common approach uses specialized modeling languages, like VRML [8], X3D [21], or COLLADA [27].

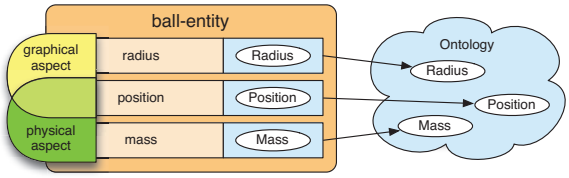

Figure 1: A ball entity consisting of three state variables, semantically linked to an ontology by means of grounded symbols. The state variable containing the ball's position is shared by the graphical and the physical aspect.

A magnitude of tools has been developed to overcome the complexity of such formats. However, expert knowledge still is required when special features are to be used [6]. This lead to the insight that a semantic description of VEs can provide a useful interface to the VE for users [28] and developers [38]. The integration of semantic descriptions, forming a KRL, eases multiple aspects of VE descriptions: Natural language description of the virtual environment [9], conceptual modeling approaches [11], decoupling of specific application content from the internals of a simulation engine [29], action representation [34], and many more.

Several projects apply ontologies to provide a KRL for VEs. Often the graphical representation is coupled with semantic information [22, 37], but also more general approaches are proposed [39]. The advantages of ontologies in the field of software engineering are pointed out by [17]. To reduce run-time overhead, often induced by accessing semantic information via an additional loosely coupled knowledge base, more recent work discusses automatic transformations of ontologies into program code [23, 43, 46].

Simulator X utilizes the Web Ontology Language (OWL) [47] to store ontologies. These are mapped into application code (see section 5) to generate unique attribute identifiers in order to handle the problem of symbol grounding [18, 34].

3 Decoupling

The presented approach is implemented using the Simulator X framework, which utilizes the actor model [19] to decouple individual components, e.g., functional building blocks like a render-, physics-, or AI-engine. Although that model facilitates the process of decoupling, the method presented in this paper does not depend thereon. The only assumption made is that the underlying abstract data model is entity centered, as suggested by [36] and [42].

3.1 Architecture elements

A central concept of Simulator X are so called *state variables*, which are used to share single values between individual components. The essential features of a state variable are reading, writing and observing its value. The latter allows to register callback functions, executed each time the variable's value is changed.

Attributes of entities can be represented by combining such variables with semantic descriptions. These are provided by means of symbols that link each variable to an asserted ontology entry. The ontology defines all objects, concepts, and their various relations as a semantic grounding for these symbols. By representing entities and their attributes using those *grounded symbols*, a direct access to the ontology and the diverse content-knowledge is provided.

Entities are conceptually subdivided into *aspects*, which are associated with components. For example, a virtual ball may have a physical and a graphical aspect. The attributes of an entity can be shared by aspects (see figure 1), e.g., the ball's position typically will be accessed by both, the physics engine and the renderer.

Since the set of all entities defines the application's global state, entities are an appropriate interface for developers and components.

Simulator X features a world interface, which serves as a registry for architecture elements by means of grounded symbols and which provides access to all entities at run-time. Furthermore, it acts as a mediator, needed by the event system described in section 4.3. The existence of a registry-service fosters decoupling of components:

As components and their functionality are developed independently, they cannot know each other in advance. Therefore, there has to be at least one mediating component, which allows to initialize handshaking processes at run-time. A specific application might require a component to access arbitrary selections of the world-state with respect to the component's aspect and the specific application logic. Both features are realized by the world interface, providing a common method and point of access.

3.2 Decoupling components

A component's functional aspect defines the entity attributes, which the component has to access. Internally, a component may still operate on its own data representations. This complicates decoupling, since either all components have to use the same types or type conversion has to be performed. The first increases coupling a lot and reduces exchangeability, as external libraries will most likely use different data types. Fortunately, because entity attributes are the only interface between components, the conversion process can be hidden by means of encapsulation. In this way a component does not depend on data types used by other components, and therefore becomes an easily exchangeable black box.

The conversion process is shown in figure 2: Component `A` updates an attribute in its local representation using the local data type. A converter automatically transforms the value into the global data type and transfers the update to the respective entity. From there the value is propagated to another component `B`, where a converter transforms it into the appropriate data type.

An important aspect of such converters is that not only data type conversion may be performed, but also the data itself can be modified. This is useful, for example, when a renderer and a physics engine employ different coordinate systems. The transformation from one system into the other may be performed by respective converters. Simulator X features a registry for different coordinate systems, allowing to perform a look-up for the required coordinate transformation.

The combination of attribute access (by means of grounded symbols) with the conversion mechanism decouples components to a great extend. The global state of the application is encoded in the set of entities, which are accessed through the world interface using grounded symbols. Each single attribute is accessible by the use of grounded symbols as well. By employing the correct symbols and utilizing the proposed conversion mechanism each component can operate completely autonomous.

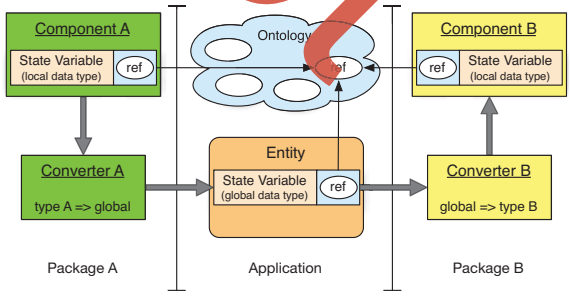

Figure 2: Two components providing converters. While the components internally use local data types, the entity contains the global type.

3.3 Variable- and entity descriptions

Local data types have to be encapsulated for the suggested automatic conversion process. This encapsulation has to be linked to the description of the respective global data type and has to include type information to ensure type-safety at compile time. In addition, a grounded symbol is required to specify the associated attribute.

Given identification of attributes using their unique semantics and their local and global data types, appropriate converters can be registered in advance and chosen and executed automatically at run-time. When an entity's attribute changes, the correct converter is selected and applied to the global-type value, to provide the component with the local data type.

Because grounded symbols are defined in an ontology, developers can easily recognize the semantics of the associated value. An AI component can (to some extend) do the same by applying reasoning mechanisms to the ontology.

Relevant parts of a respective class definition used by Simulator X are shown in listing 1: `typeinfo` holds the local data type, `base` holds the description of the global data type and `semantics` contains the grounded symbol. The method `newInstance` creates a new instance of the local data type, containing the default value specified in the ontology.

```
class SVarDescription[T, B](
  val typeinfo   : scala.reflect.ClassManifest[T],
  val base       : DescriptionBase[B],
  val semantics  : Semantics,
  newInstance    : => T
) extends ConvertibleTrait[T] with Serializable {
  val sVarIdentifier : Symbol = semantics.toSymbol
  ...
}
```

Listing 1: State variable description class from Simulator X

Converters and local type definitions have to be shipped with each component since the respective developers are the only ones to know which data types will be used locally. This is done by providing an OWL file from which variable descriptions are generated automatically (see sections 5 and 6). These can subsequently be used in the application code.

An entity is conceptually composed of different aspects. Consequently, an entity description is composed of aspects, which define the attributes of that entity (see section 4.1 for examples). Grounded symbols are used to link an aspect to the type of component for which it was designed, and to specify its semantics. A component has to recognize the latter at run-time and react appropriately. In addition, the set of associated attributes and initial values are specified by each aspect. To relate an initial value to an attribute, the concept of *state values* is introduced. These counterparts of state variables are created by means of variable descriptions, and therefore implicitly linked to all other concepts presented. Consequently, converters can automatically be applied to them as well. Figure 3 overviews the relations between all description concepts.

To instantiate an entity description a `realize` method is provided, which initializes the entity creation process described below.

3.4 The entity creation process

As an entity is shared by multiple independent components, its creation is more complex than simply calling a constructor. Each component has to provide relevant attributes and initialize its local representation on its own, in order to remain a black box. This is an easy task as long as all initial values are independent from each other, but when the initial value for an attribute depends on another one, which is provided by a different component, a build order becomes necessary.

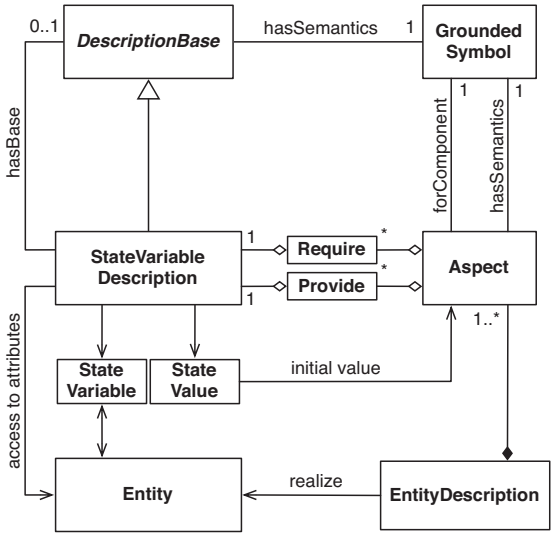

Figure 3: Relations of description concepts

Two factors affect the build process: On the one hand, a component may require an entity to be equipped with specific attributes, which the component cannot provide on its own. On the other hand, a component may depend on initial values from other components to be able to instantiate its local representation and/or provide further initial values (an example is given below). In complex cases a component may be required to provide a subset of values to other components, which subsequently calculate dependent values and pass them to the first component, before it can continue providing more values.

For this reason, aspects may contain sets of required and provided attributes, and components are given the ability to provide a set of value dependencies for each attribute or initial value associated with an aspect. Furthermore, components are given the capability to provide more attributes than specified by the respective aspect.

To instantiate the entity from a description, initial values for all aspects either have to be provided manually or must be calculated by one of the involved components. Given all the above-mentioned information a sanity check can be performed and the build order can be calculated.

The instantiation of an entity is performed in three stages: First, each involved component is passed the associated aspect. In answer thereto it provides a set of value dependencies. A value dependency contains a number of grounded symbols, which specify the values required to provide the value associated with the dependency. If, for example, a value dependency for the value `Radius` contains the grounded symbols `position` and `mass`, this means that the component requires initial values for the `Position` and `Mass` attributes to be able to provide an initial value for the `Radius` attribute. The first stage is completed by calculating the build order from the retrieved sets of value dependencies.

In the second stage each component creates its local representation of the entity, inserts the requested attributes, and sets their initial values. As described above, this stage may require the interaction of multiple components, according to the calculated build order. It is worth mentioning that, due to the automatic type conversion introduced in section 3.2, each component is independent from data types used by other components.

Finally, the entity is integrated into the simulation loop in the third stage.

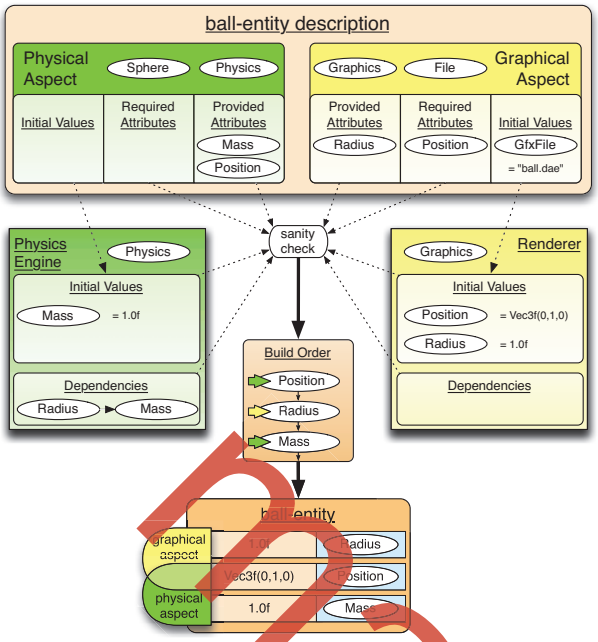

Figure 4: The entity creation process of the ball-entity from figure 1

Figure 4 depicts the entity creation process. An exemplary description of the ball-entity from figure 1 is shown. Its physical aspect is providing the mass and position attributes, whereas the graphical aspect provides its radius attribute. In addition, the graphics aspect requires the entity to have a position attribute.

The initial values for the position and radius attributes are provided indirectly by the graphical aspect (via the specified file). The value for the mass attribute, which is calculated and provided by the physics engine, depends on the radius value. A subsequent sanity check is passed, since initial values are provided for each attribute.

The following build order is calculated from the sets of value dependencies: The position attribute will be set first, since it is independent from other values. Its value will be read from the file specified in the `GfxFile` state value and then be passed to the physics engine. Since the mass attribute depends on the value of the radius attribute, the radius attribute is created first.

In order to allow entities to have subelements, entity descriptions may in turn contain entity descriptions. For this purpose a subelement-aspect, which is handled by a dedicated component, must be provided with the framework. Using this aspect, the subentity becomes an attribute of the parent entity and is assigned a grounded symbol.

In Simulator X the entity creation is started by an actor responsible for the creation process, causing all creation processes to run in parallel.

4 Integration into the Simulator X Framework

Having presented a technique to decouple the elements of RIS applications, we will now focus on the consequences for component- and application developers.

4.1 The component developers view

By means of the proposed methods components may be developed completely independent from each other. To achieve this, new components must implement the interface presented in listing 2.

The method `getDependencies` is called in the first stage of the entity creation process (see section 3.4). In the second stage `requestInitialValues` is called at least once, according to the calculated build order. In this context the parameter `given` contains a list of initial values, which were supplied before. Within this method the component has to build its local representation of the respective entity. In the third stage of the creation process the method `entityConfigComplete` is called. The method `removeFromLocalRep` is invoked each time an entity has to be removed.

```
protected def getDependencies(
      aspect : Aspect ) : Set[Dependencies]
protected def requestInitialValues(
      toProvide: Set[SVarDescription[_, _]],
      aspect: Aspect, e: Entity, given: SValList)
protected def entityConfigComplete(
      e: Entity, aspect: Aspect)
protected def removeFromLocalRep(e: Entity)
```

Listing 2: The component interface

Component developers are meant to provide (or reuse) classes that are derived from the `Aspect` class. In this way the correct mapping of grounded symbols between an aspect and the related component can be ensured. Suppose a developer specifies a physical aspect to describe a sphere. Hence, the grounded symbols `physics` (aspect type) and `sphere` (semantics) will be assigned (see listing 3). The aspect is probably specified to require or provide the attributes `Mass`, `Position`, and `Radius`, which can be specified by means of variable descriptions. A physics engine then has to recognize the `sphere` symbol and initiate appropriate actions within the methods defined in listing 2.

```
case class PhysSphere( radius : Option[Float] = None,
      position : Option[Vec3f] = None ) extends Aspect(
      Symbols.physics, Symbols.sphere ){
  def getProvidings     = Set( Mass, Position, Radius )
  def getFeatures       = Set( Mass, Position, Radius )
  def getInitialValues = {
    val values = collection.mutable.Set[SVal[_]]()
    if (position.isDefined)   retVal += Position(position)
    if (radius.isDefined)     retVal += Radius(radius)
    return values
}}
```

Listing 3: An physical aspect example

Since the symbols used throughout the application are grounded in an external ontology (see section 5), all components are able to use the same symbols. For this reason, aspects defined for one component remain applicable for others that support the same type of aspect. Even if the replacement component has a different feature set than the replaced one, an application can stay functional. In terms of the example from listing 3, the replacement component might only support `Position` and `Radius` attributes for spheres. As long as no other component depends on the `Mass` attribute (which can be verified by the sanity check of the respective entity description), the application will remain operable.

4.2 The application developers view

An application developer is rarely interested in the interaction of components with each other, but wants to focus on the design of the virtual environment. Therefore, the specification of entities and their behavior has to be decoupled from their instantiation and simulation.

Fortunately, this can also be achieved using the introduced techniques. Utilizing the set of aspects shipped with the used components, application developers only have to combine those aspects in order to create new entity descriptions. An example is given in listing 4, which results in the entity creation process from figure 4:

```
val ballDesc = new EntityDescription(
  GraphicsFile( file = "ball.dae" ),
  PhysSphere()
)

ballDesc.realize{
  (e : Entity) => println("new ball " + e + " created")
}
```

Listing 4: EntityDescription and instantiation of a virtual ball

The classes `PhysSphere` (cf. listing 3) and `GraphicsFile`, which are subclasses of the `Aspect` class, are shipped with the respective components (see section 4.1). Besides the functionality inherited from their base class, they provide a constructor with named parameters to simplify their instantiation. Furthermore, the grounded symbols, which allow the aspects to be recognized by the respective component, are hidden. The invocation of the `realize` method creates a new entity from the description, as detailed in section 3.4.

Obviously, an application developer only has to select aspects and specify attribute values in order to create entities and needs only little component-specific knowledge. Furthermore, components can easily be replaced by others, as long as they are compatible with respect to the used aspects.

Because aspects are represented in the ontology (see section 5), it is possible to define a set of basic aspects for each component type. In this way a minimal feature set can be defined, whereby basic functionality for each type of component is ensured. Component developers can extend those aspects but are also free to define completely new ones.

The feature of automatic data conversion results in the fact that the application developer only has to care about the global data types. Even if a component is replaced by another one, which is using different data types, the automatic conversion feature will hide this from the developer. For the same reasons as given for aspects in section 4.1, whole entity descriptions may be reused in different applications.

The rules of interplay within an application are often defined in the manner of if-then clauses (e.g. "if the ball collides with the table, then play a sound"). To enable the application developer to define such rules, the Simulator X framework provides event descriptions and events, as described below.

4.3 The event system

An event has features similar to an aspect. Its semantics is specified by means of a grounded symbol. Furthermore, it contains a list of state values which is used to particularize the event. As mentioned before, state values are described by state variable descriptions, for which reason they can be accessed by means of grounded symbols. Finally, a set of affected entities may be attached to an event. As with aspects, the receiving component can use the assigned grounded symbol to identify the event's type and react appropriately.

To decouple components and applications, event requests have to be mediated (e.g. by a world interface). The mediating component has to handle registrations of event providers and event handlers. After a handshaking phase, events can be passed in a peer-to-peer manner. In this context it does not matter which participant registers first, as event providers and event handlers can be stored and matched each time a new registration request is received.

Each part of an application can register for the respective type of event by sending an event description to the mediating component. Similar to variable descriptions, event descriptions and events are linked by sharing the same grounded symbol. As suggested in [42], a filtering function may be added to an event description to simplify the specification of related events.

If, for example, a physics component provides events of the type `collision`, this information is stored by the mediator. An application developer can request those events from the mediator, as shown in listing 5. The used methods are implemented by the abstract `EventHandler` class, provided with the Simulator X framework. The mediator will answer to the request with a list, containing all collision-event providers. The `EventHandler` implementation will automatically contact each of those components and register itself for the reception of events. After this procedure, the application will receive collision events from the physics engine. In listing 5 each received event is printed to the console.

```
//Within a subclass of EventHandler:
requestEvent( EventDescriptions.collision )
override def handleEvent(e: Event) { e.type match {
  case Events.collision => println(
       e.affectedEntities.mkString(" and ")+"collided" )
} }
```

Listing 5: Application registering for collision events

Events should only be used for incidents that do not reflect states. Every incident that has a duration can be realized with the help of state variables and their observe functionality. This is reasonable, because each "start-xy" event would require the implementation of a "stop-xy" event, making the code less maintainable. Sending multiple events for the duration of the incident is disadvantageous as well, as it would prevent the possibility to distinguish between two events of the same type, which take place almost simultaneously.

5 The OWL Ontology

In the previous sections we introduced a method to reduce the coupling of RIS components to the agreement on a set of symbols and associated (global) data types. However, it has to be ensured that those symbols are used and interpreted equally by each developer. As mentioned before, this corresponds to the problem of symbol grounding [18].

In order to solve this problem, those symbols and data types must be generally accessible. In addition, they have to be integrated into a taxonomy to disambiguate their meaning. These requirements bring techniques into focus, which were developed in context of the semantic web [4]. From those, the representation by means of the Web Ontology Language (OWL) [47] was chosen for multiple reasons:

First of all, it is reasonable to employ an external data format in order to obtain independence of a specific programming language. This is achieved by defining transformations from OWL into program code. Next, the choice of a standardized, widely accepted knowledge representation language offers the opportunity to utilize external resources. There exist a large number of OWL ontologies on the web, which can be employed to semantically annotate the attributes and entities used in the developed application (see section 5.3). Besides the valuable fact that AI components can benefit from those annotations, sophisticated queries on the set of entities are rendered possible. Especially when interaction with the virtual environment includes natural language input the application of such a KRL is beneficial [9]. Furthermore, it allows to define rules that have to be observed when inserting new information. Compliance with those rules can be checked automatically by the application of a reasoner. Finally, domain experts can model virtual environments without being forced to use a low-level scripting language [6]. By the use of OWL editors (like Protégé [40]) the need for programming skills can be reduced even more.

5.1 Structure

The ontology is split into three layers, which are distributed over multiple files. The structure of these layers is shown in figure 5:

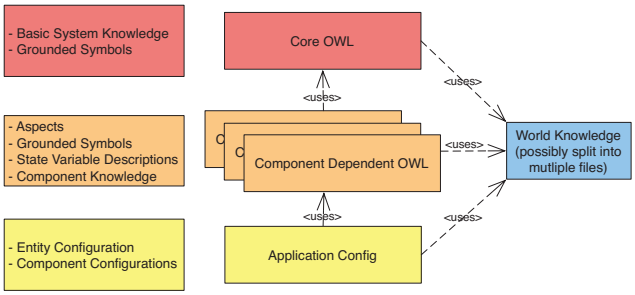

Figure 5: The proposed ontology file structure

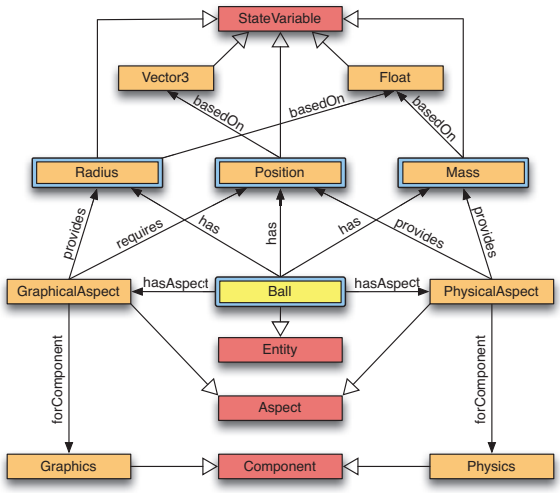

Figure 6: OWL representation of the ball entity from previous examples. The coloring reflects the belonging to the layers shown in figure 5. Concepts having a double border are suggested to be taken from an external ontology.

The architecture elements of the framework and basic classes, individuals, and properties, which are used within dependent files, are defined in the core layer. The respective files are shipped with the framework and are almost never subject to changes.

Files from the second layer are provided by the group of component developers: At least one OWL file is shipped with each component. Therein definitions from the core ontology are utilized to define symbols, aspects, and state variable descriptions that are used by the respective component. In addition, component specific knowledge may be contained in these files. It is important that all components use the same grounded symbols to ensure exchangeability. Therefore, the concepts (which are represented by grounded symbols) must be imported from external ontologies, which contain the *world knowledge* (see section 5.3).

The third layer is composed of files which are provided by the group of application developers. They contain descriptions of entities and configurations of components for specific applications. For this reason, definitions have to be imported from the OWL files which are shipped with the components that are used by the respective application.

5.2 Representation of Architecture Elements

Symbols, data types (in form of global and local variable descriptions), entity descriptions, as well as templates for aspects are represented in the ontology. The classes and object properties that are defined in the core ontology, are shown in figure 7. Besides, the OWL data properties `hasDataType` and `hasConstructor` (not shown in the figure) are defined.

The `SimulatorX_Concept` class is the base of all concepts that are transformed into program code. Grounded symbols are generated on the basis of the Internationalized Resource Identifier (IRI) [12] that is associated with each of its subclasses. In this manner an unambiguous assignment of ontology concepts and grounded symbols is obtained.

Grounded symbols for component types are generated from the subclasses of the `Component` class (e.g. from the classes `Graphics` and `Physics` in figure 6).

State variable descriptions are generated from OWL individuals whose type is a subclass of the `StateVariable` class. As mentioned before, these descriptions contain a grounded symbol as well as variable descriptions for the local and the base data type. The variable descriptions are specified using the `basedOn` and `hasDataType` properties, whereas the grounded symbol is provided by the IRI of the individual's type. Obviously, each component developer can specify an own data type for every concept, since classes can have multiple individuals. In Simulator X the default constructor (see listing 1) is defined by means of the `hasConstructor` property.

As described in section 3.3, an entity description is composed of aspects. Templates for aspects can be defined by asserting a subclass of `Aspect` and adding `provides`, `requires`, and `forComponent` property expressions.

An entity description can be specified by creating a subclass of `Entity` and specifying `hasAspect` property expressions that reference subclasses of the `Aspect` class.

The OWL representation of the ball entity, which was used in previous examples, is depicted in figure 6. For reasons of clarity the `hasDataType` and `hasConstructor` relations are not shown.

5.3 World Knowledge

As mentioned in the beginning, all developers have to use the same repository of symbols to ensure interoperability between all architecture elements. Therefore, either a central repository has to be created or an external ontology providing those symbols has to be used. The advantages of using an external ontology are the lesser effort necessary to gather symbol identifiers, and the fact that (in most cases) additional information is already linked to those symbols, which can be reused.

For example, the concepts `Position`, `Radius`, and `Mass`, shown in figure 6, are likely to be found in an external ontology. To use foreign definitions, the external ontology has to be imported, an `rdfs:subclassOf` relation between the foreign class and the `StateVariable` class has to be specified, and a `hasDataType` relation defining the global data type must be asserted. In this way, external information is preserved and can be used to infer new information on the respective object. Of course, files containing further world knowledge can be created and imported as well.

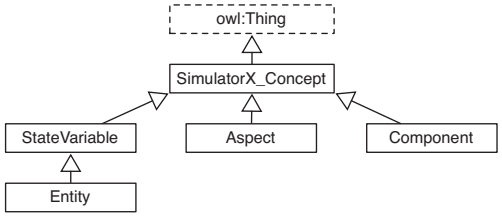

Figure 7: OWL classes defined in the Simulator X core ontology

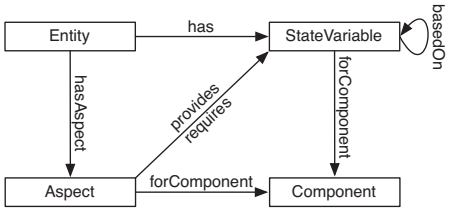

Figure 8: Object properties defined in the Simulator X core ontology

6 Code Generation

While the previous sections presented the concepts and techniques to decouple components, and thus to separate the work of different groups of developers, this section provides suggestions for the realization of those methods.

A problem related to the use of grounded symbols is the fact that errors like mistyped symbol names cannot be detected at compile time. Therefore, grounded symbols should not be represented by strings but instead stored in variables. As this would force a developer to define those symbols at least twice (in the ontology and in the program code), we generate such variables, since mistyped variable names can be detected at compile time.

To avoid further programming-related difficulties, like the risk of misused grounded symbols, it is reasonable to generate most of the code referencing those symbols as well. In particular this means that architecture elements, which are represented in the ontology, should be transformed into program code.

6.1 Implementation Details

In Simulator X the ontology is accessed using the OWL API [20]. All architecture elements that are represented in OWL files are transformed into Simulator X code. For this purpose, a set of predefined classes is provided. All relevant information can be provided by means of their constructors (see the `SVarDescription` class in listing 1 for an example). Therefore, generated classes do only have to be derived and necessary parameters have to be filled in. The predefined classes and their relationships are illustrated in the UML diagram shown in figure 3.

All generated objects are named according to the associated classes in the ontology. The `GroundedSymbol` class does not depend on other elements. It stores the identifier of a grounded symbol and is used to establish the link between the program and the ontology. Each subclass of the `VariableDescription` class needs to provide three parameters: A (local) data type, a semantic description, and an (optional) base class, which is also represented by a `VariableDescription` instance. As all of those parameters are generated, the implementation is straightforward.

The generation of aspects is more complex: An aspect has to be linked with its associated component. It also has to contain a grounded symbol, to represent its semantics. In addition, it can specify the attributes which an entity having this aspect must contain. This is done by means of an arbitrary number of `Provide` and `Require` instances, each of which carries exactly one `VariableDescription`. Currently, initial values are not stored in the ontology, for which reason only templates for aspects can be generated. For example, a physical aspect which dictates all derived aspects to have at least a position, a mass, and a shape attribute could be defined. A component developer then may create subclasses of such a template and define constructors which take the initial values as parameters. It is planned to store initial values in the ontology and generate complete aspects in the future.

To generate entity descriptions, the set of attributes, which must be provided by the aspects composing the respective entity, has to be specified. This is achieved by means of the `has` relation (see figure 8), which relates an `Entity` class to a `StateVariable` class. The aspects associated with the described entity are checked to provide all required attributes within the sanity check mentioned in section 3.4. If the check is passed, the entity is created as described. If one or more attributes are not provided by the given aspects, a revealing error message is produced.

With the help of these transformations the use of grounded symbols is hidden for the most part, whereby the risk of accidental misuse is reduced.

7 Conclusion and Future Work

We presented a technique to decouple the process of developing RIS components and applications. In this context entities provide an interface for developers and components, since the application's global state is encoded in the set of their attributes as a whole. On the technical side we use converters to decouple components, which are automatically applied to isolate the data types that are used by single components. In this way, independence of data types, as requested by [33], is obtained. Our approach reduces coupling to the agreement on global data types and symbols, which are used to access the attributes of an entity.

An ontology is utilized to overcome the related problem of symbol grounding. The concepts defined therein are transformed into code of a given target language, to reduce the risk of a mix up of symbols. The ontology also contains descriptions of basic architecture elements and entities. By transforming these descriptions into code as well, the description of VEs can be separated from the programming task. Furthermore, the use of grounded symbols can be hidden in most instances, reducing the risk of confusion of symbols even more.

Since the above-mentioned transformations are performed at compile time, and the comparison of symbols is equivalent to the comparison of references, the presented technique does hardly affect the application's performance at all. Look-ups of attributes and converters have to be performed only once per attribute and entity to store a local reference, which is subsequently used.

The presented approach was already adopted in [14], showing its practicability in a more complex scenario. Since state variables can contain an arbitrary data type, our approach was even used in combination with a skeletal animation component. However, the more specialized the contained data becomes, the smaller the set of components supporting that data will be.

Future work will address the integration of AI methods to permit querying the application's state. The modification and creation of tools to edit the ontology will be a relevant task as well. Currently a plugin for the Protégé editor, facilitating the process of symbol-, aspect-, and entity-definition, is being developed. Furthermore, a fourth layer shall be added to the ontology structure to store and restore an application's state.

Acknowledgements

Supported by the German Federal Ministry of Education and Research, program *Ingenieur*Nachwuchs, project SIRIS (#17N4409)

References

[1] J. Allard, J.-D. Lesage, and B. Raffin. Modularity for Large Virtual Reality Applications. *Presence: Teleoperators and Virtual Environments*, 19(2):142–162, April 2010.

[2] R. Aylett and M. Luck. Applying Artificial Intelligence to Virtual Reality: Intelligent Virtual Environments. *Applied Artificial Intelligence*, 14(1):3–32, 2000.

[3] A. Bachmann, M. Kunde, M. Litz, and A. Schreiber. Advances in Generalization and Decoupling of Software Parts in a Scientific Simulation Workflow System. In *ADVCOMP 2010, The Fourth International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 34–38, 2010.

[4] T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web: a new form of web content that is meaningful to computers will unleash a revolution of new possibilities. *Scientific American*, 284(5):34–43, 2001.

[5] A. D. Bierbaum, C. Just, P. Hartling, K. Meinert, A. Baker, and C. Cruz-Neira. VR Juggler: A Virtual Platform for Virtual Reality Application Development. In *IEEE Virtual Reality 2001 conference proceedings*, pages 89–96, Yokohama, Japan, 2001. IEEE Press.

[6] W. Bille, O. De Troyer, F. Kleinermann, B. Pellens, and R. Romero. Using Ontologies to Build Virtual Worlds for the Web. In *Proceedings of the IADIS International Conference WWW/Internet 2004 (ICWI2004)*, volume I, pages 683–690, Madrid, Spain, 2004.

[7] R. Blach, J. Landauer, A. Rösch, and A. Simon. A highly flexible virtual reality system. *Future Generation Computer Systems*, 14(3):167–178, 1998.

[8] R. Carey, G. Bell, and C. Marrin. ISO/IEC 14772-1:1997 virtual reality modeling language (VRML). Technical report, The VRML Consortium Incorporated, 1997.

[9] M. Cavazza. High-level interpretation in dynamic virtual environments, 1998.

[10] T. Colburn and G. Shute. Decoupling as a fundamental value of computer science. *Minds and Machines*, pages 1–19, 2011.

[11] K. Coninx, O. De Troyer, C. Raymaekers, and F. Kleinermann. VR-DeMo: a tool-supported approach facilitating flexible development of virtual environments using conceptual modelling. *Proc. of Virtual Concept 2006*, 2006.

[12] M. Duerst and M. Suignard. Internationalized Resource Identifiers (IRIs). RFC 3957, Internet Engineering Task Force, 2005.

[13] P. Figureoa. InTml: Main Concepts, Examples. and Initial Lessons. In Latoschik et al. [30], pages 3–6.

[14] M. Fischbach, D. Wiebusch, A. Giebler-Schubert, M. E. Latoschik, S. Rehfeld, and H. Tramberend. SiXton's curse - Simulator X demonstration. In *Virtual Reality Conference (VR), 2011 IEEE*, pages 253–256. IEEE, 2011.

[15] E. Frécon and M. Stenius. DIVE: a scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3):91–100, 1998.

[16] C. Greenhalgh and S. Benford. Massive: a distributed virtual reality system incorporating spatial trading. In *ICDCS '95: Proceedings of the 15th International Conference on Distributed Computing Systems*, page 27, Washington, DC, USA, 1995. IEEE Computer Society.

[17] H. Happel and S. Seedorf. Applications of ontologies in software engineering. In *Proc. of Workshop on Sematic Web Enabled Software Engineering"(SWESE) on the ISWC*, 2006.

[18] S. Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346, 1990.

[19] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI'73: Proceedings of the 3rd international joint conference on Artificial intelligence*, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[20] M. Horridge and S. Bechhofer. The OWL API: A Java API for Working with OWL 2 Ontologies. In *6th OWL Experienced and Directions Workshop*, Chantilly, Virginia, 2009.

[21] ISO/IEC, JTC 1/SC 24. X3D abstract. Technical Report 19775-1:2004, ISO/IEC, 2004.

[22] E. Kalogerakis, S. Christodoulakis, and N. Moumoutzis. Coupling ontologies with graphics content for knowledge driven visualization. In *Proceedings of the IEEE VR2006*, pages 43–50, 2006.

[23] A. Kalyanpur, D. Pastor, S. Battle, and J. Padget. Automatic mapping of OWL ontologies into Java. In *Proceedings of the International Conference on Software Engineering & Knowledge Engineering (SEKE)*, 2004.

[24] P. Kapahnke, P. Liedtke, S. Nesbigall, S. Warwas, and M. Klusch. ISReal: an open platform for semantic-based 3D simulations in the 3D internet. *The Semantic Web–ISWC 2010*, pages 161–176, 2010.

[25] A. Kapolka, D. McGregor, and M. Capps. A unified component framework for dynamically extensible virtual environments. In *Fourth ACM International Conference on Collaborative Virtual Environments*, 2002.

[26] J. Kelso and L. E. Arsenault. Diverse: A framework for building extensible and reconfigurable device independent virtual environments. In *IEEE Virtual Reality*, pages 183–190, 2002.

[27] Khronos Group. COLLADA 1.5.0 Specification. http://www.khronos.org/files/collada_spec_1_5.pdf, Oct 2008.

[28] F. Kleinermann, O. De Troyer, H. Mansouri, R. Romero, B. Pellens, and W. Bille. Designing semantic virtual reality applications. In *Proceedings of the 2nd INTUITION International Workshop, Senlis, France*, pages 5–10, 2005.

[29] M. E. Latoschik and C. Fröhlich. Towards intelligent VR: Multi-layered semantic reflection for Intelligent Virtual Environments. In *Proceedings of the Graphics and Applications GRAPP 2007*, pages 249–259, Barcelona, Spain, 2007.

[30] M. E. Latoschik, D. Reiners, R. Blach, P. Figueroa, and R. Dachselt, editors. *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), proceedings of the IEEE Virtual Reality 2008 workshop*. Shaker Verlag, 2008.

[31] M. E. Latoschik and H. Tramberend. Engineering Realtime Interactive Systems: Coupling & Cohesion of Architecture Mechanisms. In T. Kuhlen, S. Coquillart, and V. Interrante, editors, *Proceedings of the Joint Virtual Reality Conference of Euro VR – EGVE – VEC*, EG Symposium Proceedings, pages 25–28, 2010.

[32] M. E. Latoschik and H. Tramberend. Simulator X: A Scalable and Concurrent Software Platform for Intelligent Realtime Interactive Systems. In *Proceedings of the IEEE VR 2011*, 2011.

[33] S. Limet, S. Robert, and A. Turki. Flowvr-sciviz: a component-based framework for interactive scientific visualization. In *CBHPC '09: Proceedings of the 2009 Workshop on Component-Based High Performance Computing*, pages 1–9, New York, NY, USA, 2009. ACM.

[34] J.-L. Lugrin and M. Cavazza. Making Sense of Virtual Environments: Action Representation, Grounding and Common Sense. In *Proceedings of the Intelligent User Interfaces IUI'07*, 2007.

[35] B. Macintyre and S. Feiner. A distributed 3d graphics library. In *In Computer Graphics (Proc. ACM SIGGRAPH '98), Annual Conference Series*, pages 361–370, 1998.

[36] F. Mannuß, A. Hinkenjann, and J. Maiero. From scene graph centered to entity centered virtual environments. In Latoschik et al. [30], pages 37–40.

[37] J. Martinez and C. Mata. A basic semantic common level for virtual environments. *International Journal of Virtual Reality*, 5(3):25–32, 2006.

[38] J. Otto. Towards semantic virtual environments. October 2003.

[39] F. Pittarello and A. De Faveri. Semantic description of 3D environments: a proposal based on web standards. *Proceedings of the eleventh international conference on 3D web technology*, pages 85–95, 2006.

[40] Protege. The Protégé Ontology Editor and Knowledge Acquisition System. http://protege.stanford.edu/, 2007.

[41] C. Shaw, M. Green, J. Liang, and Y. Sun. Decoupled simulation in virtual reality with the MR toolkit. *ACM Transactions on Information Systems (TOIS)*, 11(3):287–317, 1993.

[42] A. Steed. Some useful abstractions for re-usable virtual environment platforms. In Latoschik et al. [30], pages 33–36.

[43] G. Stevenson and S. Dobson. Sapphire: Generating Java Runtime Artefacts from OWL Ontologies. In *Proceedings of the Third International Workshop on Ontology-Driven Information Systems Engineering*, June 2011.

[44] R. M. Taylor, II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. Vrpn: a device-independent, network-transparent vr peripheral system. In *VRST '01: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61, New York, NY, USA, 2001. ACM.

[45] H. Tramberend. Avocado: A distributed virtual reality framework. In *Proceedings of the 1999 IEEE Conference on Virtual Reality (VR-99)*, Los Alamito, CA, 1999.

[46] M. Völkel and Y. Sure. RDFReactor-from ontologies to programmatic data access. In *Poster Proceedings of the Fourth International Semantic Web Conference*, 2005.

[47] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview. Technical report, W3C, October 2009.